in: Proc. 9th Doctoral Consortium of the 14th Int. Conf. on Advanced Information Systems Engineering (CAiSE'2002), Toronto, May, 2002

Towards a modular, object-relational schema design

Wolfgang Mahnke

University of Kaiserslautern
P.O.Box 3049, 67653 Kaiserslautern, Germany
mahnke@informatik.uni-kl.de

Abstract. A modular or even component-based design is state-of-the-art in application development. But both object-relational database management systems and the object-relational standard SQL:1999 lack proper abilities supporting a modular schema design – the prerequisite for a component-based schema design. Nevertheless, the high effort required for an object-relational schema design strictly demands for a modular schema design. This paper describes the challenges of such a schema design and gives a vision of how a modular, object-relational schema design should look like.
Keywords: schema design, ORDBMS, modularity, CBSE.

1 Motivation

„Modularity is the key to achieving the aims of reusability and extensibility" stated Bertrand Meyer in 1988 [13]. Nowadays, software is developed even in a component-based way [1,5]. The idea behind this effort is to develop parts of the functionality independently of the context in which it is supposed to be used and later on to build large systems by the combination of precasted components. This leads to a high degree of component reuse and, in turn, to a faster development of large systems.

Regarding database schema design, there was no effort of doing it in a component-based way for a long time. This is due to the fact that the relational data model is very simple since there are just a few different kinds of schema elements (mainly table and view definitions). But with the upcoming object-relational database systems this situation has quite changed [16]. First, there are many new concepts in the object-relational data model like user-defined types (UDTs), triggers, table hierarchies, etc., which lead to a much more complex schema design. Second, parts of the application logic can be integrated into the database system using user-defined routines (UDRs). Corresponding with a higher effort required for the schema design, the need to reuse (parts of) the schema grows. Furthermore, dividing parts of the schema into loosely coupled units leads to a higher maintainability of the schema. However, the dependencies between the schema elements have to be kept under control.

Whereas component models like J2EE [17] and DCOM [4] offer an environment for a component-based application development, there are no such mechanisms in ORDBMSs (see Section 3 for details).

In the following section, the research question is illustrated by an example of an object-relational schema design. The related work is introduced in Section 3. Afterwards, Section 4 presents a framework for a modular, object-relational schema design by describing the different kinds of schema modules and their relationships.

One kind of schema module – the schema component – is discussed in detail. Then we take a look at the relationships between the schema components and the components outside the database. Section 5 finishes the paper with a conclusion and an outlook to future work.

2 Research Question

The research question regarded in this paper can be phrased as:

„*How should a modular, object-relational schema design look like?*“

To obtain a deeper motivation for the need of a modular schema design, we give a brief overview of an object-relational schema design without modularity. In this example, the schema encompasses different semantic units. But the standard SQL:1999 offers no possibility to aggregate the different schema elements (tables, UDRs, UDTs, etc.) to schema modules. The example will point out different dependencies between the schema elements and, in turn, dependencies between the semantic units. A classification of the dependencies can be taken from [12]. In a modular schema design, all dependencies between schema modules have to be made explicit. Implicit dependencies have to be prohibited.

The example is used to denote all dependency types and illustrate them in an explicit way. It is a simplified excerpt derived from the SFB-501-Reuse Repository[1] [6,9].

The Reuse Repository is designed to support all phases of a reuse process and the accompanying improvement cycle of the Quality Improvement Paradigm [3] by providing adequate functionality. To gain more experience with the new object-relational technology we have chosen the, as we call it, *extreme extending* (X^2) approach, i. e., almost everything has been implemented by using the extensibility infrastructure of the ORDBMS. Thus, X^2 means that not only the entire application logic runs within the DB server, but also major parts of the presentation layer (GUI) reside within the DB server, because HTML pages used for user interaction are dynamically generated within the DBS. In this context, we do not want to describe the functionality of the Reuse Repository in detail. Briefly summarized, its main functionality is to manage experience data and support a similarity based search on such data.

Due to the fact that our ORDBMS does not support a reference type, we have implemented a UDT called `ObjectID`. An `ObjectID` value serves as a unique identifier and stores information about the storage location (table) and the type of the object. In addition to the `ObjectID`, there is a typed table called `root_ta` (see Fig. 1) of the type `root_ty` including an `ObjectID` as primary key and some triggers maintaining the `ObjectID` (e.g., keeping values unique). All tables using the `ObjectID` inherit from `root_ta`. Together these schema elements build a semantic unit providing an object identifier.

Another aspect of the Reuse Repository is related to user management, which is simplified here as a typed table `user_ta` of the type `user_ty`. Because a user needs an `ObjectID`, the type inherits from `root_ty` (1) and the table from

---

1. The SFB-501-Reuse Repository is part of the subproject A3 *“Supporting Software Engineering Processes by Object-Relational Database Technology“* of the Sonderforschungsbereich 501, *“Development of Large Systems by Generic Methods*“, funded by the German Science Foundation.

root_ta (2). This kind of dependency is called *refinement dependency*.

The experience management plays a central role in our Reuse Repository. In our example, experience data is stored in the form of characterization vectors (CV). Similar to the user, a CV should have an ObjectID. Therefore, the typed table cv_ta inherits from root_ta (3) and its type cv_ty from root_ty (4). Because the build-in types of the ORDBMS are not expressive enough to represent a CV, we used the UDT html (5) of the Informix WebBlade [8]. The use of data types of other semantic units establishes *structural dependencies*. In the CV, its creator has to be recorded, which is realized with a foreign key relationship to user_ta (6). The corresponding dependency is called *reference dependency*.

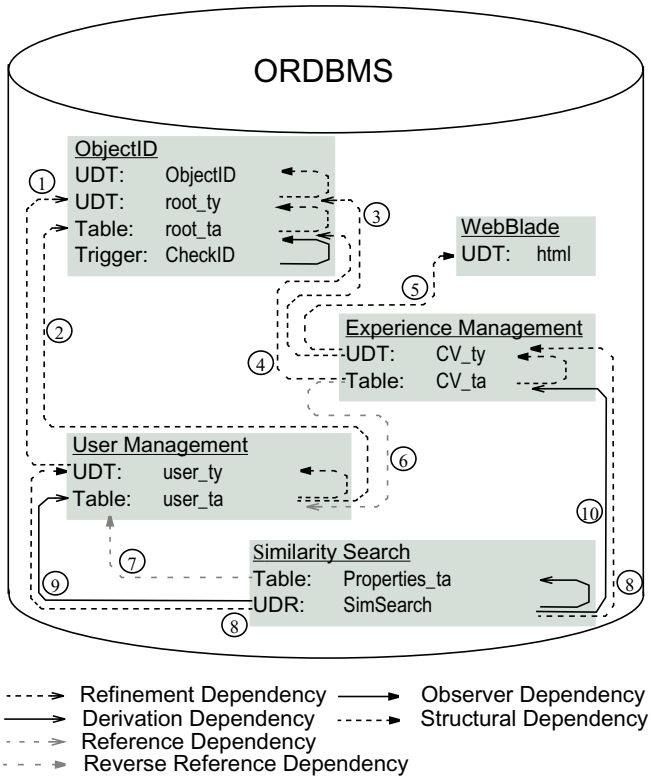

Figure 1. Reuse Repository schema

To retrieve experience data we have implemented a similarity-based search by a UDR called SimSearch. The measures of similarity can be specified by parameters. These parameters are stored in a table called properties_ta. To apply a specific similarity function, each user refers to a set of parameters, where the same set of parameters (e.g., the default set) can be used by several users. This relationship is represented by a foreign key between user_ta and properties_ta. Hence, the foreign key attribute is included in the user_ta (7). Although the dependency is caused by the similarity search, the foreign key is realized in the referenced semantic unit, the user management. Such a kind of dependency is called *reverse reference dependency*. Calling SimSearch, occurrences of a user_ty and of a cv_ty as comparison instance have to be specified as parameters (8). SimSearch selects the parameters for the similarity function by reading the user_ta and the property_ta (9). Afterwards a query is evaluated on cv_ta (10) and the results are returned ordered by the similarity value. Evaluating a query on tables of other semantic units leads to a *derivation dependency*.

Although we have observed a modular schema design in the Reuse Repository, the modular structure can not be seen at the schema level. It can only be found in the documentation. Because the implicit, hard-to-find dependencies are not made explicit, reuse of single schema modules is impossible. Even if we would have an explicit module structure, some kinds of dependencies would prevent the reuse of schema modules and, therefore, have to be avoided. For example, the *reverse reference dependency* changes the structure of the referenced module. Therefore, the ref-

erenced module can not be reused in another context.

Consequently, for a modular schema design we demand:

- semantically interrelated schema elements must be managed within schema modules;
- implicit dependencies between schema elements have to be made explicit at the module level;
- schema modules and their relationships have to be managed as part of the schema and disallowed dependencies have to be prevented.

So far, the example only deals with schema design. Later on, in Section 4.3 we will regard the relationship between schema modules and components outside the ORDBMS. But first, we will briefly consider related work in Section 3, and present a framework for a modular, object-relational schema design in Section 4.

3 Related Work

Related work concerning object-relational schema design can be divided into two categories: the object-relational standard SQL:1999 and additional mechanisms offered by the ORDBMS vendors.

SQL:1999

In the object-relational standard SQL:1999 [2], the only way of grouping schema elements is using the concept of server modules, which are not sufficient for our purposes. They can only contain UDRs and temporary tables, but can not be nested to compose new server modules, have no interface concept and no relationships can be defined between different server modules (see [12] for details).

ORDBMS vendors

The ORDBMS vendors offer some mechanisms to aggregate schema elements called DataBlades [7] or cartridges [14]. But those are just administrative units to (de-)install a group of schema elements at a time. The schema elements are simply mapped to an unstructured, flat schema, thereby losing their inherent structure (see [12] for details).

Another mechanism to group mainly UDRs offered by Oracle is called package [15]. But packages can not include all kinds of schema elements, e.g., table definitions, trigger definitions, etc. are not supported, they can not be nested and no relationships can be defined between different packages.

Summarizing it can be said that there are a few approaches to group schema elements, but all of them are not sufficient for a modular, object-relational schema design.

4 A Framework for a Modular, Object-Relational Schema Design

In this section, we present a framework for a modular, object-relational schema design. We will discuss the need of different schema modules in Section 4.1, take a closer look at the most important kind of schema modules – the schema component in Section 4.2, and regard the relationship between schema modules and components outside the ORDBMS in Section 4.3.

4.1 Different kinds of schema modules

Using our sample scenario we have identified different kinds of schema modules. Such modules differ in the sort of schema elements they can hold, in the way they are instantiated, and in the kinds of relationships among them. We have identified three kinds of modules: schema components, schema packages and schema frameworks (see [11]).

A *schema component* can exist independently of other schema modules. The User Management, the Experience Management, and the Similarity Search are examples of schema components. Schema components are defined once and can be instantiated several times. For example, the User Management may be needed more than once in a single schema to manage different kinds of users for different applications using the same schema. The definition of a schema component only contains definitions of the schema elements included. If it is instantiated, the corresponding schema elements are instantiated within the namespace of the component. Each component instance has its own namespace, in order to avoid name conflicts. All kinds of schema elements can be included in a schema component.

A *schema package* serves to provide schema element definitions (like UDT or UDR definitions) to other schema modules. For example, the schema package WebBlade offers the `html` data type. A schema package can not be instantiated, but the schema elements defined in it are instantiated whenever it is used. These elements belong to the namespace of the component that uses the package. Because a schema package is always local to a schema component, it can not contain any sort of schema elements dealing with a global view, for example roles, user rights, and assertions.

Another kind of schema module, offering a precasted framework, is called *schema framework*. It has to be enhanced with concrete properties. Therefore, it can not be instantiated by itself and has to be completed by a schema component. The ObjectID is an example of a schema framework. Schema frameworks can be instantiated in two ways; either as a private instantiation only for a single schema component in the namespace of the component or to be used by many components. In the latter case, the schema framework has its own namespace and many components can complete the framework, like the ObjectID in the example. Comparable to schema components it may contain all kinds of schema elements.

Depending on its type, a schema module may use other modules in different ways. Using other schema modules leads to dependencies at the schema element layer. To keep these dependencies under control, all relationships between the modules have to be specified explicitly, and only dependencies according to these relationships are permitted. The different relationship types are illustrated in Fig. 2.

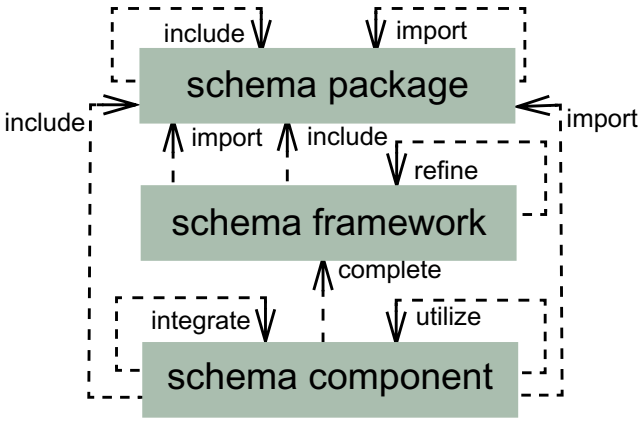

Figure 2. Relationships between schema

A schema package can only use other schema packages. If schema package P_1 *includes* P_2, the schema elements of P_2 are copied into the namespace of P_1. If P_1 *imports* P_2, P_1 can use the schema elements of P_2, but they are still in the namespace of P_2. Hence, the schema elements are only referenced.

Schema frameworks may *include* or *import* schema packages. These relationship types have the same semantics as described above. A schema framework F_1 can *refine* another schema framework F_2. In this case, F_2 is enhanced by some properties, but the resulting F_1 is still no self-contained schema component. The schema elements of F_2 are copied into the namespace of F_1.

Comparable to the other schema modules, a schema component may *include* or *import* a schema package. When instantiating a schema component C_1, the schema elements of the included P_1 are instantiated within the namespace of C_1, whereas the schema elements of the imported P_2 use the namespace of P_2 within the namespace of C_1. A schema component C_1 can *complete* a schema framework F_1. This relationship type is parameterized. A private completion means that F_1 is exclusively completed by C_1 and the schema elements of F_1 are included into the namespace of C_1. In the case of a public completion, F_1 can be completed by many schema components. Therefore, F_1 uses its own namespace outside the namespace of C_1. Finally, C_1 may *integrate* or *utilize* another schema component C_2. Integrating means that C_2 is a private component of C_1 and only accessible via C_1, whereas utilizing means that C_2 is an independent schema component that can be accessed using its own namespace.

Lets take a more concrete look at schema modules. Unlike current SQL-DDL statements, where each statement execution leads to an immediate instantiation of a schema element, schema modules first have to be defined and can be instantiated later on. This is necessary because some schema modules (schema packages and schema frameworks) can not be instantiated by themselves and schema modules may be instantiated several times in different namespaces. Therefore, a schema module definition itself should only hold definitions of schema elements.

4.2 Schema components

The definition of a schema component consists of three parts: the schema element definitions belonging to the schema component, the interfaces implemented by the schema component and the interfaces needed by the schema component. In the following, we will take a closer look at these parts.

4.2.1 Schema element definitions

The main part of a schema component definition consists of the schema element definitions that belong to the schema component. In Fig. 3, the schema element definitions are illustrated in the internal structure as polygons and possible *apply modes* as triangles on the polygons. An apply mode describes how a schema element can be used. For example, a table has apply modes for selecting, inserting, defining foreign keys on it, etc. Inside the component definition each schema element can use the other schema elements in any possible way (lines between the schema element definitions). Furthermore, the metadata of the schema element definitions are part of the internal structure. These are stored in an enhanced information schema corresponding to the information schema of a database schema.

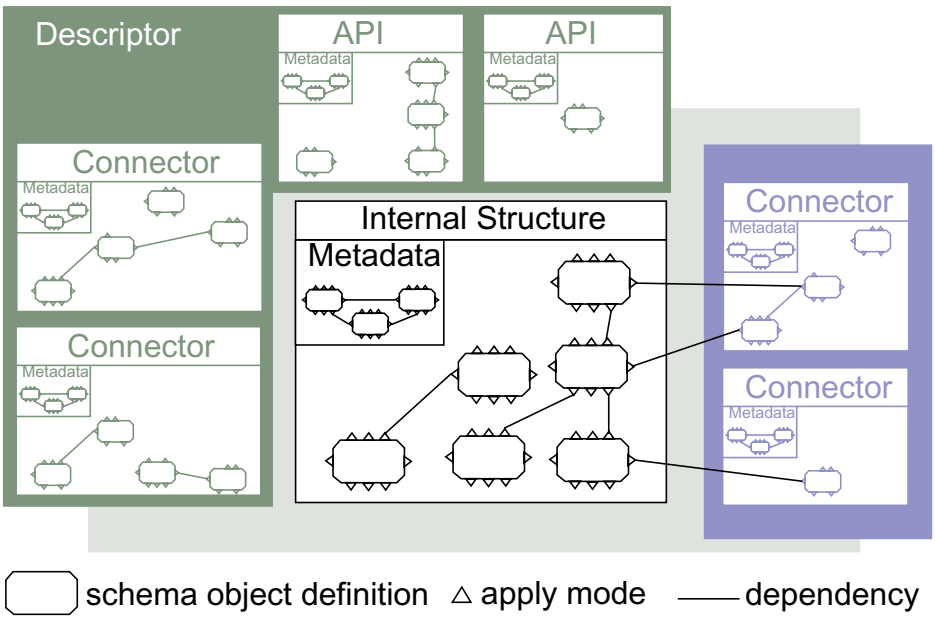

Figure 3. Illustration of a Schema Component Definition

4.2.2 Interfaces of a schema component

An interface of a schema module can be used for different purposes. One kind of interface defines how other schema modules can interact with the schema module. Such an interface is called *connector*. Another kind of interface defines how the applications outside the database may access the schema module, which is called *API* (application programming interface). Using these interfaces (connectors and APIs) we can encapsulate the internal structures and implementation details of schema modules. The schema modules can only be accessed via the interfaces and schema modules implementing the same interfaces can be exchanged independently of their internal structure.

An interface in the context of a database schema consists of more than just the signatures of UDRs. An API may include all schema elements accessible from outside the database like tables and UDRs. A connector can hold all schema elements to be used by other schema elements. This excludes assertions, triggers, etc., which can not be used by other schema elements (but nevertheless may themselves use other schema elements), but includes tables, UDTs, etc. The apply modes of the schema element definitions in APIs and connectors differ. It does not make sense to allow a foreign key definition on a table in an API. In a connector, however, this is a suitable choice. An interface does not contain the whole definition of a schema element, e.g., a UDR just needs to offer its signature. Each interface offers the metadata of its content. The metadata schema corresponds to the enhanced information schema including information about the apply modes.

Defining an interface may lead to an incorrect interface. Assume an interface I_1 that holds the definition of a table T_1 having an attribute of a UDT U_1. If the apply mode of T_1 includes a SELECT and U_1 is not part of I_1, then there is an inconsistency in the interface. Therefore, we distinguish two kinds of interfaces, an explicit and an implicit one. An explicit interface only holds the definitions that are explic-

itly specified. This can lead to the inconsistencies mentioned above and would raise an error when defining the interface. Using an implicit interface, dependencies like the one mentioned above are resolved by putting all needed elements implicitly into the interface. In summary, an explicit interface may lead to errors, but the developer keeps control over each element definition inside the interface. An implicit interface avoids errors, but may lead to an interface that was not intended. To support the preferences of different designers, both kinds of interfaces should be offered.

Note that there are definitions and instances of schema elements and schema modules, but only interfaces. This is because interfaces can not be instantiated and therefore are only defined.

Using other schema modules

A schema component may need some other schema modules. For example, the schema component Experience Management needs a type `html`. In the definition of the schema component we do not want to fix which schema module is used for this purpose. Doing this would lead to the situation that we can not exchange the schema module that offers the `html` type. Instead we define a connector that is needed by the schema component and any schema module that implements the connector can be used when instantiating the schema component. Fig. 3 illustrates that two connectors are used (the connectors at the right hand side). The schema element definitions in these connectors have restricted apply modes. All accessible apply modes can be used inside the schema component definition (see the lines in Fig. 3). The metadata of the connectors are part of the metadata of the internal structure – possibly using its own namespace. This depends on the kind of relationship to the connector.

Providing interfaces

A schema component can offer one or many connectors describing in which way the component can be used by other components. Furthermore, it can provide one or more APIs describing in which way it can be used from applications outside the database. Both kinds of interfaces hold an excerpt of the schema element definitions, this can include parts of the connectors used. The apply mode can be limited (see Fig. 3, connectors at the left side and APIs on top of the schema component). For example, an API can hold a table definition, but only allow SELECT statements on the table. INSERTs are not allowed directly and encapsulated in a UDR. Both kinds of interfaces hold metadata describing how they can be used. So, all interfaces of a schema component together can be seen as the descriptor of the component.

4.2.3 Instantiating a schema component

To instantiate a schema component, a name for the schema component has to be provided. This name is used as the namespace for the component elements in the schema and has to be unique inside the schema. If the schema component definition contains connectors which are used by the schema component, it has to be specified which schema modules are used for that purpose. Depending on the kind of relationship to the connector, an already existing schema module has to be identified (e.g., if another schema component is utilized) or the name of the schema module definition has to be specified (e.g., if a schema component is integrated it has to be newly created).

4.3 Relationships between schema components and application components

Our Reuse Repository uses an approach integrating almost everything into the ORDBMS (the extreme extending approach (X^2) [9]), but nevertheless we have implemented some components working outside the ORDBMS. In the following, we extend our example to point out how the interplay between the schema components and the components outside the ORDBMS looks like.

The Reuse Repository has a web-based interface and therefore offers a web form to enter the data of a new CV. In addition, we have implemented a way to hand over new CVs as XML documents. The corresponding Java-based application runs outside the ORDBMS (see Fig. 4). If an XML document is handed to the application (1), we use the JAXP parser provided by Sun [18] to parse the document and validate it against an XML-schema document. This XML-schema document is provided by the component XML2CV inside our application, more exactly by the subcomponent XMLSchemaProvider. The XMLSchemaProvider has a counterpart in the Reuse Repository, a schema component called XMLSchemaStorage, which is used to store the XML-schema documents (3). For this scenario, it is important to know that there are different CV types in a type hierarchy, which are not shown in Fig. 1. The attributes of the CVs may change and the hierarchy may be extended. Hence, we have a single XML-schema document for each CV type and the XML-schema documents can become invalid if the CV types change. Therefore, the XMLSchemaProvider has to check if its document is still valid by calling a function of the component CVAccess (4). If the XML-schema document is still valid, it can be delivered to the Validator (5). Otherwise, a new one has to be generated. In that case, XMLSchemaProvider calls a UDR of XMLSchemaStorage. The UDR gets information from the Experience Management (a) and another schema component called MapCV2XML (b), holding information about how to map the attributes of the CV to the XML elements and attributes. The generated XML-schema document is stored in XMLSchemaStorage and delivered to the Validator via XMLSchemaProvider (5). If the XML document is valid w.r.t. the given XML-schema document, it is transformed into a DOM representation (6). The DOM representation is used as input for CVAccess (7). Together with the information of MapCV2XML (8) and information about the CV type provided by Experience Management (9), it

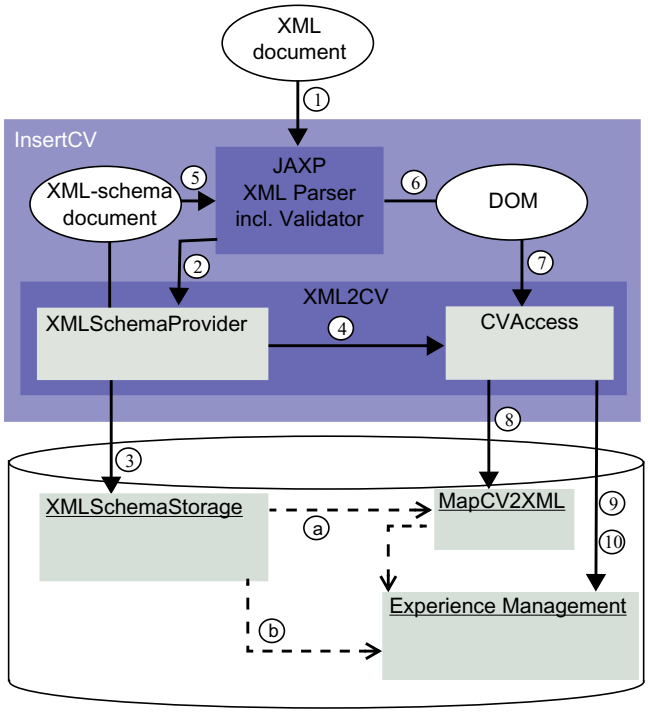

Figure 4. Applications and the Reuse Repository

generates an SQL query to insert the CV (10).

Analyzing this scenario reveals that the two schema components MapCV2XML and XMLSchemaStorage belong to the application. More exactly they belong to the application component XML2CV. If we divide the component into subcomponents, we can attach the component XMLSchemaProvider at the application layer to the schema component XMLSchemaStorage at the database layer. Hence, supporting schema components does not only mean to permit a vertical composition of components, but also a horizontal composition of components. The composed component offers interfaces at the application layer and interfaces at the database layer. CVAccess and MapCV2XML build a new composed component and its interfaces are used at the application layer and at the schema layer. In our example, we only considered vertical access between the composed components, but horizontal access is possible, too. For example, XMLSchemaProvider may access MapCV2XML directly and MapCV2XML may call a function of the XMLSchemaProvider to generate a new XML-schema document, if the mapping has changed.

5 Conclusions and Outlook

The example of the Reuse Repository schema has shown some weaknesses of an object-relational schema design and motivated the need for a modular, object-relational schema design. Different kinds of schema modules where introduced: schema packages, schema frameworks, and schema components. The latter has been discussed in detail, in particular the different interfaces of schema components. Afterwards the relationships between application components and schema components have been viewed on the basis of an extended example of the Reuse Repository.

Using the introduced framework for a modular, object-relational schema design leads to a clearer, better manageable schema design and admits the reuse of schema modules. In addition, schema components enable the possibility to build new components by composing schema and application components and thereby simplifying the composition of application components w.r.t. their persistent data.

As further work, we will refine the specification of the different schema modules, interfaces and relationships between the schema modules. We will use a metamodelling approach to specify the different module classes and the relationships to the schema elements, interfaces, etc. based on the UML. This approach seems to be suitable w.r.t. the complexity of the problem. With the semi-formal specification, we want to propose a language extension of SQL:1999 supporting the needed modular schema design. Furthermore, we want to develop a system based on an existing ORDBMS accepting the proposed SQL:1999 extension for a modular schema design.

Other open issues in the context of our work are:

- What operations on schema modules are needed and how do they work?
- Can we use the „design by contract“ approach of Bertrand Meyer [13] for UDRs in the interfaces? This would lead to a better understanding of the interface and more stable code in the database.
- What happens to persistent data when exchanging a schema component after it is instantiated? Changing schema components may lead to an easier way to handle schema evolution.

- Are there measures for a good modular schema design? Can those measures be calculated and help the designer to provide a good modular schema design?
- Do we need parameterized modules (like C++ templates)?

Our experience in different projects [6,9,10] using object-relational technology has shown that without a modular schema design handling ORDBMSs is extremely hard, in particular if application logic is integrated into the ORDBMS.

Literature

[1] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Peach, B., Wust, J., Zettel, J.: Component-Based Product Line Engineering with UML. Addison Wesley, 2002.

[2] ANSI/ISO/IEC 9075-2-1999: Database Languages - SQL - Part 2: Foundation (SQL/Foundation). American National Standard Institute, Inc., 1999.

[3] Basili, V. R., Caldiera, G., Rombach, H. D.: Experience Factory. In J. J. Marciniak (ed), Encyclopedia of Software Engineering, Volume 1, John Wiley & Sons, 1994, pp. 469-476.

[4] Brown, N., Kindel, C.: Distributed Component Object Model Protocol - DCOM/1.0. Microsoft Corp., 1998.

[5] D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML - The Catalysis Approach. Addison Wesley, 1998.

[6] Feldmann, R.L., Geppert, B., Mahnke, W., Ritter, N., Rößler, F.: An ORDBMS-based Reuse Repository Supporting the Quality Improvement Paradigm - Exemplified by the SDL-Pattern Approach. TOOLS USA 2000, Santa Barbara, CA, July, 2000, pp. 125-136.

[7] IBM: IBM Informix DataBlade Module Development Overview, Version 4.0. IBM Corporation. August, 2001.

[8] IBM: IBM Informix Web DataBlade Module - Application Developer's Guide, Version 4.13. IBM Corporation. December, 2001.

[9] Mahnke, W., Ritter, N.: The ORDB-based SFB-501-Reuse Repository. VIII. International Conference on Extending Database Technology (EDBT 2002), Demo Presentation Session, Prague, 2002, pp. 745-748.

[10] Mahnke, W., Ritter, N., Steiert, H.-P.: Towards Generating Object-Relational Software Engineering Repositories. 8. GI-Fachtagung „Datenbanken in Büro, Technik und Wissenschaft" (BTW'99), Freiburg, March 1999, pp. 251-270.

[11] Mahnke, W., Steiert, H.-P.: It is time to apply the principles of component-based software engineering to the design of object-relational databases! (in German) GI/OCG-Jahrestagung Informatik 2001, Wien, September 2001, pp. 823-828.

[12] Mahnke, W., Steiert, H.-P.: Modularity in ORDBMSs - A new Challenge. 13. Workshop „Grundlagen von Datenbanken", GI-FG 2.5.1, Magdeburg, June 2001, pp. 83-87.

[13] Meyer, B.: Object-oriented Software Construction. Prentice Hall, 1988.

[14] Oracle: Oracle 9i Data Cartridge Development's Guide. Oracle Cooperation, June, 2001.

[15] Oracle: Oracle 9i Supplied PL/SQL Packages and Types Reference. Oracle Corporation, June, 2001.

[16] Stonebraker, M., Brown, M.: Object-Relational DBMSs - Tracking the Next Great Wave. Morgan Kaufmann, 1999.

[17] Sun Microsystems: Enterprise JavaBeans Specification, Version 2.0, Sun Microsystems, Inc., April 2001.

[18] Sun Microsystems: Java API for XML Processing, Version 1.1, Sun Microsystems, Inc., February 2001.